A TYPE SYSTEM FOR OBJECT MODELS

Jonathan Edwards,
Daniel Jackson &
Emina Torlak
Computer Science & AI Lab · MIT

FSE · Newport Beach, CA · Nov 3, 2004

what's an object model? a type?

object model
› first-order constraints
› over set/relation structure

typical uses
› data modelling (ER, UML)
› runtime assertions (OCL)
› policy, ontology, etc (RDF)
› behavioural modelling (Z, Alloy)

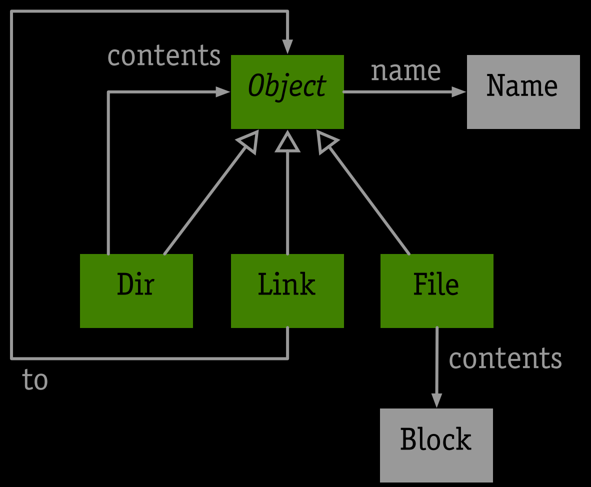
why types?
› find errors at 'compile time'

do such simple languages really need complex type systems?

problem

initial goals
› catch 'subtype errors'
› resolve overloading of relations

existing approaches
› don't support subtypes (Z)
› allow undecidable types (PVS)
› adopt approach like Java's (OCL)



-- *no directory points to itself*
no d: Dir | d = d.to

-- no file has empty contents
no f: File | f.contents = none

solution

key ideas
› untyped semantics
› type error = irrelevant expression
› resolution = all but one resolvent irrelevant

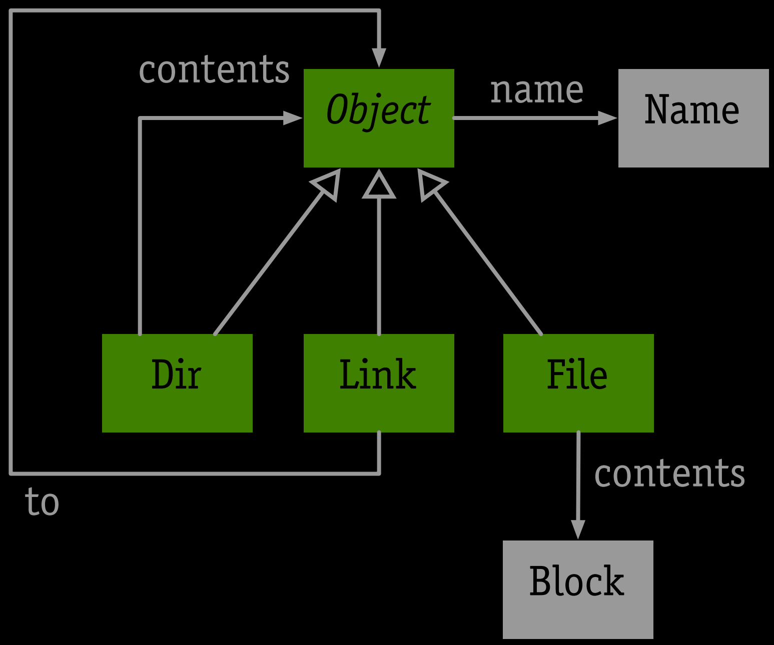
outcomes
› simpler language, no casts
› no false alarms
› very flexible resolution

examples: simple cases

-- *every object has a name*
all o: Object | some n: Name | o.name = n

-- *every block has a name*
all b: Block | some n: Name | b.name = n
type error: b.name = Ø, so it's irrelevant

-- *no directory points to itself*
no d: Dir | d = d.to
subtype error: d.to = Ø, so it's irrelevant



examples: look ma, no casts!

-- root directory contains only directories

some root: Dir | root.contents in Dir

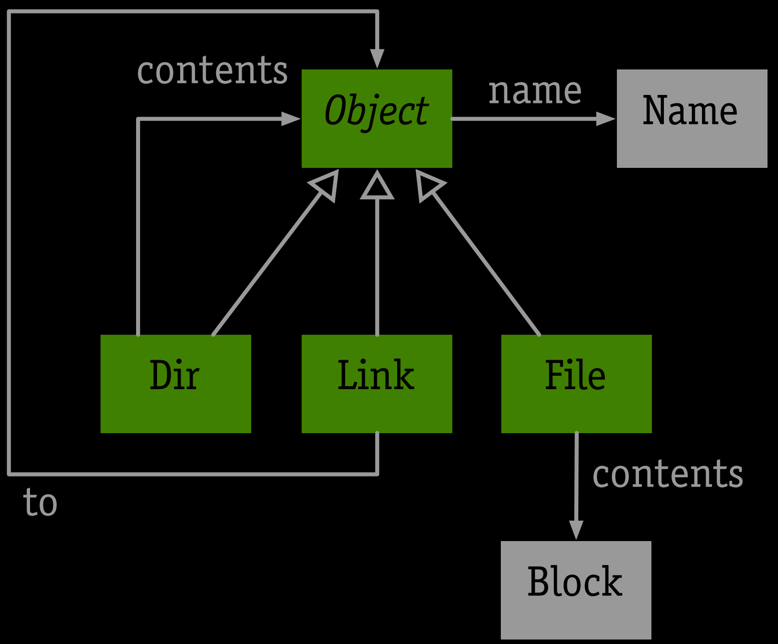
OK, even though root.contents may include non-Dir

(Dir) root.contents in Dir ?? -- cast is pointless

-- no directory pointed to by link of descendant

no d: Dir | d in d.^contents.to

OK, even though contents may yield non-Link

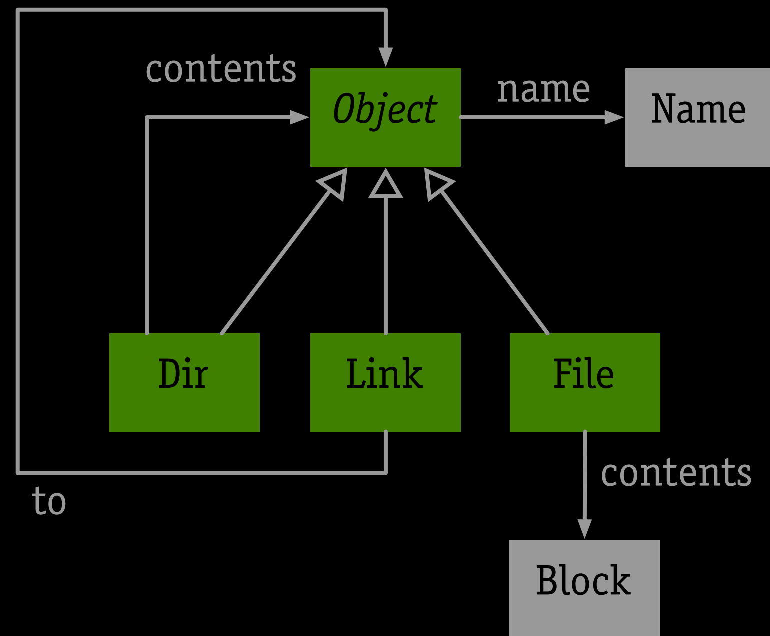


examples: resolution

-- *every object has some contents*
no o: Object | o.contents = none
contents is ambiguous

-- *no file contains itself*
no f: File | f in f.contents
f.contents is irrelevant; can replace by Ø

-- *no object contains itself*
no o: Object | o in o.contents
resolved OK; uses full context



syntax

formula ::= elemFormula | compFormula | quantFormula

elemFormula ::= expr **in** expr | expr **=** expr

compFormula ::= not formula | formula and formula

quantFormula ::= (all | no) var **:** expr **|** formula

expr ::= rel | var | none | expr binop expr | unop expr

binop ::= + | & | - | . | ->

unop ::= ~ | ^

semantics

M: Formula, Binding → Boolean

$M[\text{not } f]b = \neg M[f]b$

$M[\text{all } x: e \mid f]b =$
$\quad \wedge \{M[f]\ (b \oplus x \mapsto v) \mid v \subseteq E[e]b \wedge \#v=1\}$

$M[p \text{ in } q]b = E[p]b \subseteq E[q]b$

E: Expression, Binding → RelationValue

$E[p+q]b = E[p]b \cup E[q]b$

$E[p\,.\,q]b = \{\langle p_1,\ldots,p_{n-1}, q_2,\ldots,q_m\rangle \mid$
$\quad \langle p_1,\ldots,p_n\rangle \in E[p]b \wedge \langle q_1,\ldots,q_m\rangle \in E[q]b \wedge p_n = q_1\}$

$E[p \text{->} q]b = \{\langle p_1,\ldots,p_n, q_1,\ldots,q_m\rangle \mid$
$\quad \langle p_1,\ldots,p_n\rangle \in E[p]b \wedge \langle q_1,\ldots,q_m\rangle \in E[q]b\}$

$E[\hat{}p]b = \{\langle x, y\rangle \mid$
$\quad \exists p_1,\ldots p_n \mid \langle x, p_1\rangle, \langle p_1, p_2\rangle, \ldots \langle p_n, y\rangle \in E[p]b\}$

variables: $E[x]b = b(x)$

relations: $E[r]b = \cup \{b(r_i) \mid r_i \text{ has name } r\}$

declarations

› in semantics, just constraints

Dir in Object, Object in Dir + Link + File, name in Object -> Name

› in type system, gives subtype structure

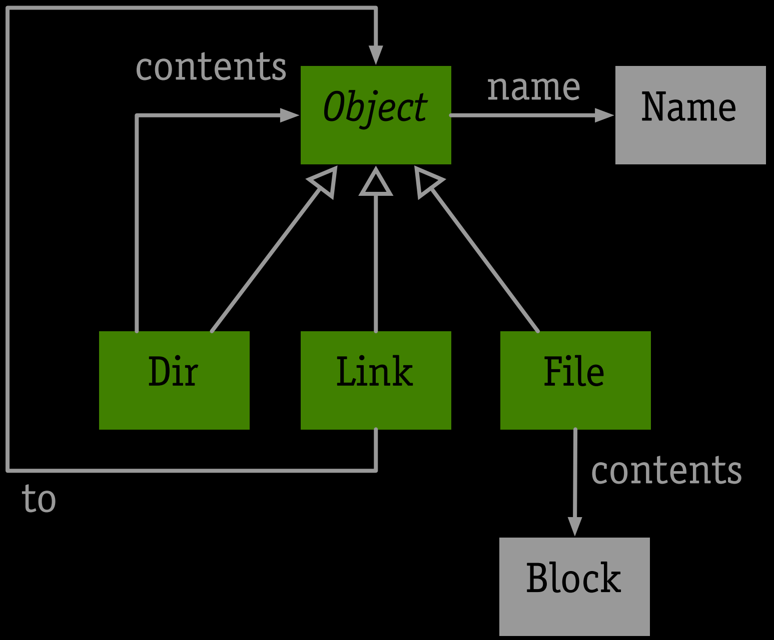
```
abstract sig Object {
  name: Name}

sig Dir extends Object {
  contents: set Object}

sig File extends Object {
  contents: set Block}

sig Link extends Object {
  to: Object}

sig Name, Block {}
```



types

basic type is leaf of hierarchy

Dir, Link, File, Block, Name

relational type is sum of products

$contents_{File}$: File -> Block

Object: Dir + Link + File

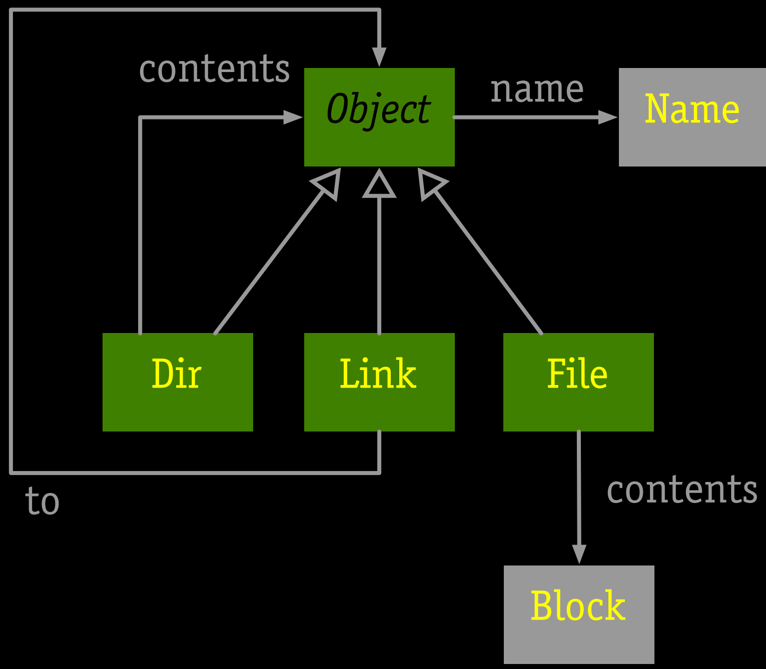
name: Dir->Name + Link->Name + File->Name

… ie, a relation!

$contents_{File}$: {(File,Block)}

Object: {(Dir),(Link),(File)}

name: {(Dir,Name),(Link,Name),(File,Name)}



consequences

› no subtype comparisons

› compute types with relational operators

› requires mixed-arity calculus

bounding type *e* : *t*

approximates expression value

› with a relational type

› computed using relational operators

› report error if empty

example

-- no directory is linked to or contains itself

no d: Dir | d in ($d.contents_{Dir}$ + d.to)

$d.contents_{Dir}$: {(Dir)} . {(Dir,Dir),(Dir,Link),(Dir,File)} = {(Dir), (Link), (File)}

d.to : {(Dir)} . {(Link,Dir),(Link,Link),(Link,File)} = Ø

so d.to is ill-typed

syntactic fragility

instead of

`no d: Dir | d in (d.contents`$_{Dir}$` + d.to)`

consider the equivalent formula

`no d: Dir | d in d.(contents`$_{Dir}$` + to)`

now there is no type error

› no subexpression with type Ø

problem is that to is irrelevant

› even though not Ø, can replace by Ø

relevance types *e :: t*

approximates portion of expression value
› that is relevant to the enclosing formula
› similar computation, but top-down
› report error if empty

example

no d: Dir | d in d.($contents_{Dir}$ + to)

because d has type {(Dir)}

d.($contents_{Dir}$ + to) :: {(Dir)}

because ($contents_{Dir}$ + to) has type

{(Dir,Dir),(Dir,Link),(Dir,File), (Link,Dir),(Link,Link),(Link,File)},

$contents_{Dir}$ + to :: {(Dir,Dir)}

because to has type {(Link,Dir),(Link,Link),(Link,File)}

to :: Ø

soundness: bounding

bounding types

› key property

$$e \subseteq \mathrm{type}(e)$$

› sample rule

$$\frac{p : P,\ q : Q}{p + q : P \cup Q}$$

soundness: relevance

bounding types

› key property

$$e \subseteq \text{type}(e)$$

› sample rule

$$\frac{p : P,\ q : Q}{p + q : P \cup Q}$$

relevance types

› key property

$$F\,[e \cap \text{type}(e)/e] = F[e]$$

› sample rule

$$\frac{p + q :: T,\ \ p : P}{p :: P \cap T}$$

hoare's formulation

for each function *f* of the language, assign

› a covariant bounding function f^{+}

› a contravariant relevance function f^{-}

such that

$$e \cap E = e \Rightarrow f(e) \cap f^{+}(E) = f(e)$$

$$f(e) \cap F = f(e) \Rightarrow f(e \cap f^{-}(F)) = f(e)$$

resolving overloading

to resolve overloading
› semantically, relation just denotes union
› resolves if all but one resolvent is irrelevant

example

no d: Dir | d in d.contents

is short for

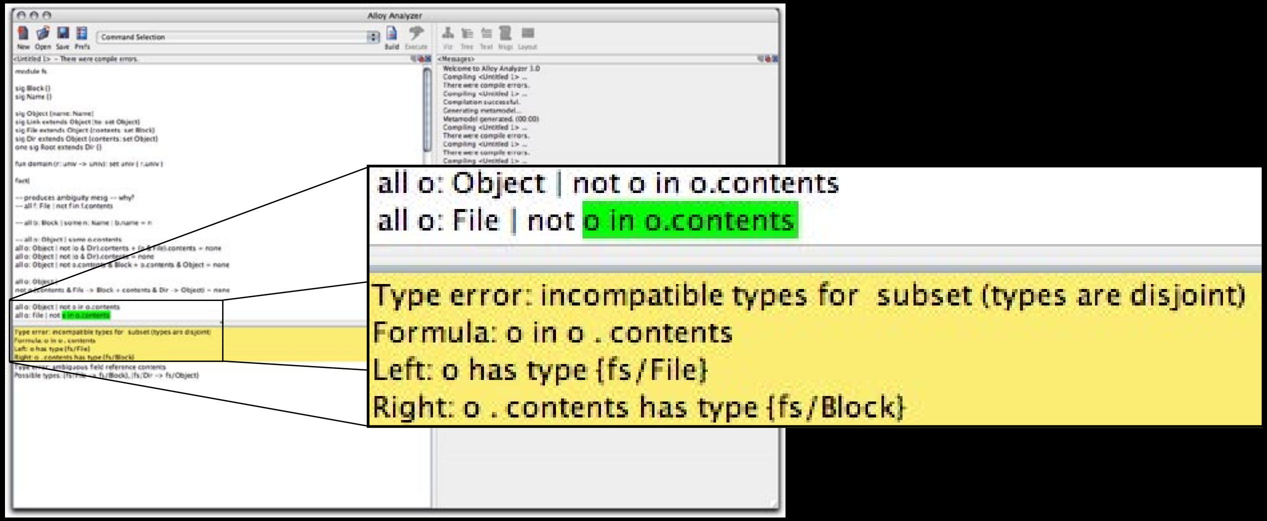
no d: Dir | d in d.($contents_{Dir}$ + $contents_{File}$)

$contents_{File}$ will be found to be irrelevant

nice consequences
› no additional mechanism needed
› consistent with untyped semantics
› not dependent on syntactic form (eg, x.r)

realization in alloy 3.0

- union types as byproduct
- ‘univ’ + functions = subtype polymorphism
- atomization exploits subtypes in analysis
- parametric polymorphism too



realization in alloy 3.0

- union types as byproduct
- 'univ' + functions = subtype polymorphism
- atomization exploits subtypes in analysis
- parametric polymorphism too
- see alloy.mit.edu

```
one sig Null {}
sig LinkedList {header: Entry}
    {all e: header.*next |
        no e.(next+prev) & Null and e.prev.next = e}
sig Entry { next, prev: Entry + Null, element:
univ }
```

comparison: alloy 2

problems

› overloading only for 'top level types'

› no real namespace for subsignature

› ad hoc rules for resolution

› no detection of subtype errors

```
sig Object {
  name: Name}

sig Dir extends Object {
  contentsD: set Object}

sig File extends Object {
  contentsF: set Block}

sig Link extends Object {
  to: Object}

sig Name, Block {}

fact { no d: Dir | d in d.to }
```

comparison: Z

problems
› no overloading (except for schemas?)
› no subtypes or union types
› schemas can't be used for classification

$$
\begin{array}{l}
[Obj, Name, Block] \\
\\
FileSystem = [\\
\quad Object, File, Dir, Link: \mathbb{P}\, Obj \\
\quad contentsF: Obj \leftrightarrow Block \\
\quad to: Obj \nrightarrow Obj \\
\quad | \\
\quad contentsF \in File \leftrightarrow Block \\
\quad to \in Link \rightarrow Object \\
\quad \forall d: Dir \bullet \neg\, d = (to\ d) \\
\quad]
\end{array}
$$

comparison: UML

problems

- overloading? not clear
- complicated semantics
- casts & special type operators
- no relational operators
- casts prevent navigating over sets

Alloy

```
d.contents.to
```

OCL

```
d.contents
  ->select (oclIsTypeOf (Link))
  ->collect (oclAsType(Link).to)
```

conclusions

It may be possible to have the best of both worlds
by adding typing annotations to an untyped specification language.

--Lamport & Paulson. Should your specification language be typed? TOPLAS, 1999.

we've shown this can be done, but
› for a first-order language
› without partial functions

questions
› higher-order languages?
› applications to programs?
› basis for a programming language?